DLP Lab6 Report

Shu Kai, Lin

May 2025

1 Introduction

In this lab, I implemented a Conditional Denoising Diffusion Probabilistic Model (DDPM) and conducted experiments with a Latent Diffusion Model (LDM) to generate 64×64 images conditioned on multi-label object labels. The Conditional DDPM was built using the Hugging Face diffusers library and customized with a label-aware UNet backbone, cosine noise schedule, and classifier-free guidance (CFG) to enhance generation fidelity. The LDM, on the other hand, performs denoising in a compressed latent space using a pretrained VAE from diffusers to encode and decode images, significantly reducing computational cost while maintaining image quality. Both models were trained on the provided ICLEVR dataset and evaluated using a pretrained ResNet18-based classifier. The best configuration achieved an evaluation score of 0.8, demonstrating the effectiveness of both pixel-space and latent-space diffusion approaches.

2 Implementation Details

2.1 DDPM

Objective:
The goal of the Conditional Denoising Diffusion Probabilistic Model (DDPM) is to model the data distribution by reversing a gradual noising process. In this implementation, conditioning is done using multi-label object vectors from the ICLEVR dataset. The model learns to predict the noise added to an image at each timestep, enabling sample generation by iterative denoising. The loss function used is a simple mean squared error between the predicted and true noise:

$$\mathcal{L}_{\text{DDPM}} = \mathbb{E}_{x,y,t,\epsilon}\left[\|\epsilon - \epsilon_\theta(x_t, y, t)\|_2^2\right] \tag{1}$$

where x_t is the noisy image, y is the multi-label condition, t is the timestep, and ϵ is the true noise
UNet Architectures:
Two different UNet designs were explored to condition the noise prediction process on label embeddings:

- **Linear Projection UNet ('learn_CondUNet'):**
  In this design, the one-hot label vector is passed through a three-layer feed-forward network (with SiLU activations) to project it into a dense embedding space of dimension 64. The

resulting embedding is reshaped and broadcast to match the spatial resolution of the input latent tensor (e.g., $64 \to 64 \times 8 \times 8$) and concatenated along the channel dimension with the noisy image. The concatenated tensor is then passed into a Hugging Face 'UNet2DModel' with 4 input channels (noisy latents) + 64 label channels. This design explicitly injects label information as a spatial map and allows the model to learn rich conditioning dependencies. It performed well with low-resolution latent inputs.

```python
class learn_CondUNet(nn.Module):
    def __init__(self, n_cls, emb_dim=64):
        super().__init__()
        self.linear_projection = nn.Sequential(
            nn.Linear(n_cls, 128),
            nn.SiLU(),
            nn.Linear(128, 256),
            nn.SiLU(),
            nn.Linear(256, emb_dim),
        )
        self.unet = UNet2DModel(
            sample_size=64,
            in_channels=4 + emb_dim,
            out_channels=4,
            layers_per_block=2,
            block_out_channels=(128, 256, 512, 768),
            down_block_types=(
                "DownBlock2D",
                "AttnDownBlock2D",
                "DownBlock2D",
                "DownBlock2D"
            ),
            up_block_types=(
                "UpBlock2D",
                "UpBlock2D",
                "AttnUpBlock2D",
                "UpBlock2D"
            ),
            time_embedding_type="positional",
        )
    def forward(self, lat, t, onehot):
        emb = self.label_embedding(onehot)
        emb = emb[:, :, None, None].expand(-1, -1, lat.size(2), lat.size(3))
        x = torch.cat([lat, emb], dim=1)
        return self.unet(x, t).sample
```

- **Hugging Face Conditional UNet ('CondUnet'):**
  This version is a direct extension of Hugging Face's 'UNet2DModel'. Labels are encoded as one-hot vectors, reshaped and expanded spatially (e.g., from $B \times 24$ to $B \times 24 \times 64 \times 64$), and concatenated directly with the image input before being fed into the UNet. This model uses a deeper UNet with six blocks on both encoder and decoder sides, including attention layers, making it more expressive and suitable for high-resolution image generation. This version is more parameter-heavy but provides improved performance when trained directly in pixel space.

```python
                self.label_embedding = nn.Embedding(n_cls, embedding_label_size)
```

2.2 Noise Schedules

DDPM Noise Schedule:
For both the Conditional DDPM and LDM models, I used a cosine-based noise schedule ('squaredcos_cap_v2') provided by Hugging Face's 'DDPMScheduler'. This schedule gradually increases the noise variance across timesteps following a cosine curve, allowing smoother transitions and better performance compared to the linear beta schedule. The forward diffusion process corrupts the image x_0 into x_t by:

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon \tag{2}$$

where $\epsilon \sim \mathcal{N}(0, I)$ and $\bar{\alpha}_t$ is the cumulative product of $(1 - \beta_t)$.

During sampling, the reverse process is modeled step-by-step using the learned noise predictor $\epsilon_\theta(x_t, t, y)$:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(x_t, t, y) \right) + \sigma_t z \tag{3}$$

where $z \sim \mathcal{N}(0, I)$ and σ_t is derived from the schedule.

DDIM Sampling:
In addition to standard DDPM sampling, I implemented DDIM (Denoising Diffusion Implicit Models) sampling for the progressive image generation and CFG-enhanced sampling. DDIM follows the same training process as DDPM but enables faster and deterministic sampling by skipping timesteps using a non-Markovian update:

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}} x_0 + \sqrt{1 - \bar{\alpha}_{t-1} - \sigma_t^2} \cdot \epsilon_\theta(x_t, t, y) \tag{4}$$

This allows fewer steps (e.g., 50–200 instead of 1000) and stable visual quality during CFG-based generation. I used this method to generate intermediate denoising images for visualizing the denoising trajectory.

```
        if self.args.cfg_weight > 0:
            self.noise_scheduler = DDIMScheduler(num_train_timesteps=self.num_train_timestamps,
            ↪  beta_schedule="squaredcos_cap_v2", prediction_type="epsilon",clip_sample=True)
            self.noise_scheduler.set_timesteps(self.num_train_timestamps)
        else:
            self.noise_scheduler = DDPMScheduler(num_train_timesteps=self.num_train_timestamps,
            ↪  beta_schedule="squaredcos_cap_v2")
            self.noise_scheduler.set_timesteps(self.num_train_timestamps)
```

Deciding which to use is based on cfg weight affect or not.

2.3 Classifier-Free Guidance (CFG)

To further improve conditional generation, I implemented **Classifier-Free Guidance (CFG)** during both training and sampling. CFG helps the model focus on the condition label without needing an external classifier.

Training CFG:
During training, label dropout is used: with a fixed probability (e.g., 10%), the label input is replaced with a zero vector, effectively simulating unconditional generation. This helps the model learn both conditional and unconditional behaviors within the same architecture.

```python
    def diffusion_loss(self,img, lbl, cfg_dropout_p=0.1):
        t = torch.randint(0, self.noise_scheduler.config.num_train_timesteps-1, (img.size(0),),
        ↪  device=img.device).long()
        noise = torch.randn_like(img)
        noisy = self.noise_scheduler.add_noise(img, noise, t)
        if self.args.cfg_weight > 0:
            drop_mask = (torch.rand(img.size(0), device=img.device) < cfg_dropout_p)
            lbl[drop_mask] = 0.0
        return F.mse_loss(self.noise_predicter(noisy.to(torch.float), t, lbl.to(torch.float)), img)
```

Sampling CFG:
At inference time, CFG is performed by making two noise predictions:

$$\epsilon_{\text{final}} = \epsilon_{\text{uncond}} + w \cdot (\epsilon_{\text{cond}} - \epsilon_{\text{uncond}}) \tag{5}$$

where w is the guidance strength (CFG weight), typically between 1.5 and 3. This encourages the model to strongly follow the condition without sacrificing image quality. This approach was used in both DDPM and LDM sampling, leading to improved alignment with label conditions and higher evaluation accuracy.

```python
    def evaluate(self, epoch, split="test", cfg_weight=0.0):
        test_dataset = ICLEVERDataset(root_dir=self.args.data_dir, json_file=f"{split}.json",
        ↪  objects_json="objects.json")
        test_dataloader = DataLoader(test_dataset, batch_size=32)
        for y in test_dataloader:
            y = y.to(self.device)
            x = torch.randn(32, 3, 64, 64).to(self.device)
            B = x.shape[0]
            if(cfg_weight == 0):
                for t in tqdm(self.noise_scheduler.timesteps, desc="Sampling",
                ↪  total=self.noise_scheduler.config.num_train_timesteps, leave=False):
                    with torch.no_grad():
                        pred_noise = self.noise_predicter(x.to(torch.float), t.to(torch.float),
                        ↪  y.to(torch.float))
                    x = self.noise_scheduler.step(pred_noise, t, x).prev_sample
            else:
                # With CFG
                for t in tqdm(self.noise_scheduler.timesteps, desc="Sampling (CFG)", leave=False):
                    with torch.no_grad():
                        x_2B = torch.cat([x, x], dim=0)
                        y_2B = torch.cat([y, torch.zeros_like(y)], dim=0)
                        eps_2B = self.noise_predicter(x_2B, t, y_2B)
                        eps_c, eps_u = eps_2B.chunk(2)
                        eps = eps_u + cfg_weight * (eps_c - eps_u)
                    x = self.noise_scheduler.step(eps, t, x).prev_sample
            acc = self.eval_model.eval(images=x.detach(), labels=y)
            denormalized_x = (x.detach() / 2 + 0.5).clamp(0, 1)
            print(f"accuracy of {split}.json on epoch {epoch}: ", round(acc, 3))
            generated_grid_imgs = make_grid(denormalized_x)
            save_image(generated_grid_imgs, f"{self.svae_root}/{split}_{epoch}.jpg")
        return round(acc, 3)
```

2.4 Latent Diffusion Model (LDM)

Overview:
In this implementation, I adopted the Latent Diffusion Model (LDM) framework, where the diffusion process is performed in a compressed latent space instead of the full pixel space. This significantly reduces computational cost and memory usage while maintaining generation quality. The model consists of three key components: a pretrained VAE for encoding/decoding images, a modified UNet for latent noise prediction, and a diffusion scheduler for training and sampling.

Pretrained VAE (AutoencoderKL):
To encode and decode images in latent space, I utilized the pretrained `AutoencoderKL` model from Hugging Face's `diffusers` library, specifically `stabilityai/sd-vae-ft-mse`. This model compresses a $3 \times 64 \times 64$ RGB image into a $4 \times 8 \times 8$ latent tensor. For stability, only the mean of the latent distribution is used during training. A fixed scaling factor of 0.1821 is applied to the latent to align with the expected input range of the UNet. The decoder reconstructs images from the latent, outputting pixel values in $[-1, 1]$, which are then rescaled to $[0, 1]$ for visualization and evaluation.

```
1    vae = AutoencoderKL.from_pretrained("stabilityai/sd-vae-ft-mse").to(args.device)
2    vae.requires_grad_(False)
```

UNet Modification for Latent Input:
To accommodate the VAE's 4-channel latent representation, I modified the UNet to accept $4+24 = 28$ input channels, where 24 is the length of the one-hot label vector. The label is broadcast spatially to 8×8 and concatenated with the noisy latent tensor before being passed into the network. The UNet is based on Hugging Face's 'UNet2DModel', configured with six encoder and decoder blocks and spatial attention in the bottleneck. This architecture enables the model to effectively learn the denoising process within the compressed representation space while conditioning on semantic labels.

Classifier-Free Guidance (CFG):
To improve generation fidelity under conditional settings, I employed Classifier-Free Guidance (CFG). During training, labels are randomly dropped with a fixed probability (e.g., 10%), allowing the model to learn both conditional and unconditional behaviors. At inference time, two predictions are made—one conditional and one unconditional—and combined as:

$$\epsilon_{\text{final}} = \epsilon_{\text{uncond}} + w \cdot (\epsilon_{\text{cond}} - \epsilon_{\text{uncond}}) \tag{6}$$

where w is the CFG weight. This encourages the model to generate images that better match the given label set without relying on an external classifier.

DDIM Sampling:
To accelerate inference and enable progressive denoising visualization, I implemented the Denoising Diffusion Implicit Model (DDIM) as an alternative to standard DDPM sampling. DDIM uses the same noise prediction model and training process but deterministically maps noise back to data using non-Markovian updates, allowing sampling from a reduced number of steps. This reduces generation time while maintaining visual quality and conditional accuracy. In my implementation, DDIM is used in sample_z'cfg by adjusting the scheduler to a smaller number of timesteps (e.g., 50), which is particularly useful for visualizing the denoising trajectory in fewer frames.

3 Results & Discussion

3.1 Training Outcome

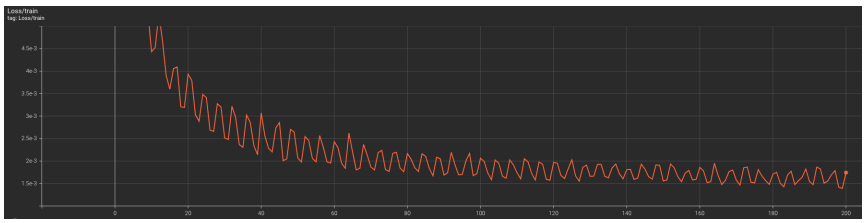


Figure 1: DDPM Training Loss

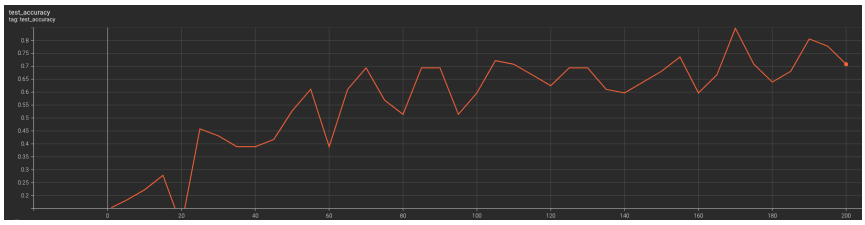


Figure 2: Test.json Accuracy

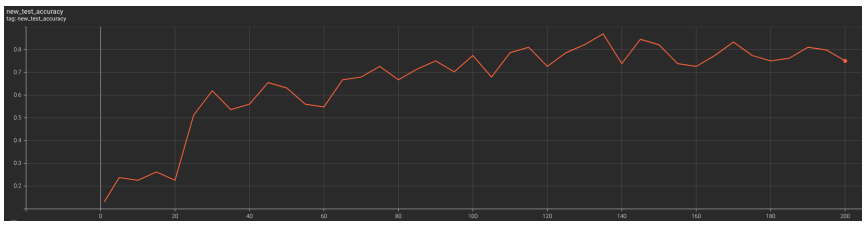


Figure 3: New_Test.json Accuracy

3.2 Synthetic Image Grids

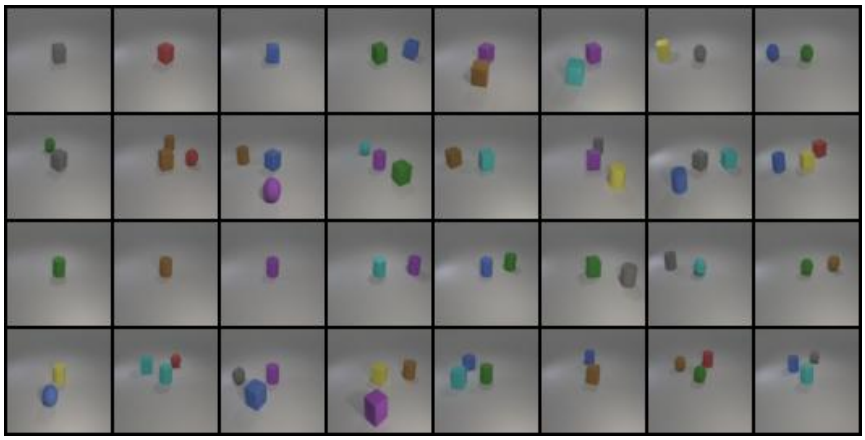


Figure 4: Test.json Results

Figure 5: New test.json Results

```
accuracy of test.json on epoch 170:  0.847
accuracy of new_test.json on epoch 170:  0.833
```

Figure 6: Accuracy Screen Shot

3.3 Denoising process image

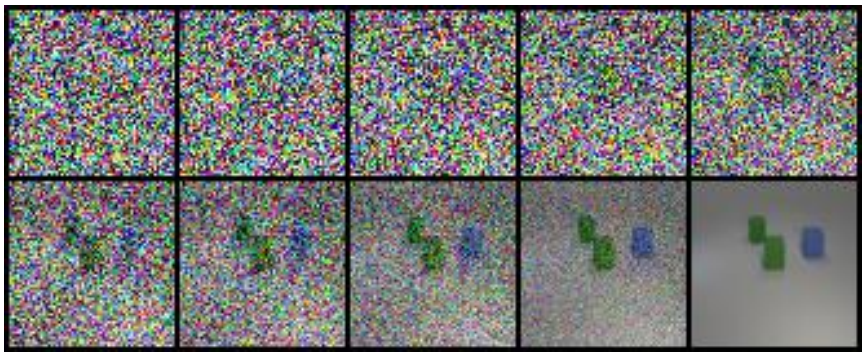


Figure 7: Denoising Process

4 Experimental Results

4.1 Using Learnable Project Layer

I use linear project layer instead of built in static embedding, however, the learnable classifier seems to mess up.

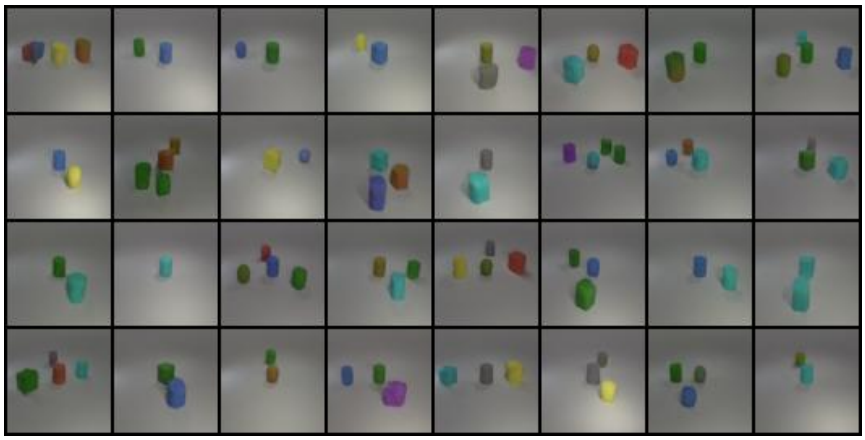


Figure 8: Learnable Unet test.json

```
accuracy of test.json on epoch 50:  0.528
accuracy of new_test.json on epoch 50:  0.631
```

Figure 10: Learnable Unet Accuracy Screen Shot

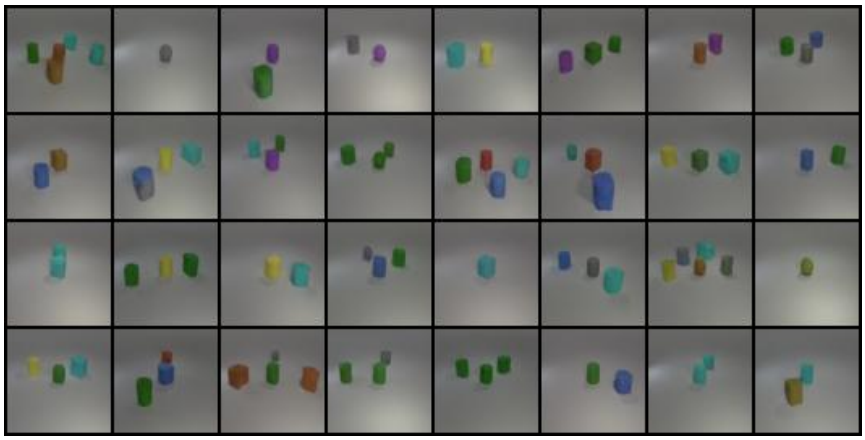


Figure 9: Learnable Unet new_test.json

4.2 Using CFG with DDPM

I test using DDPM with CFG and using DDIM in infering stage to ensure memory sufficient, yet the outcome seems terrible.

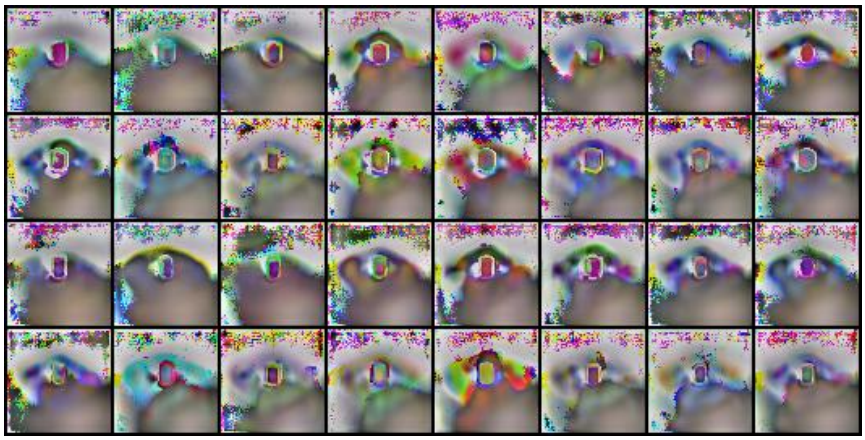


Figure 11: DDPM w CFG test.json

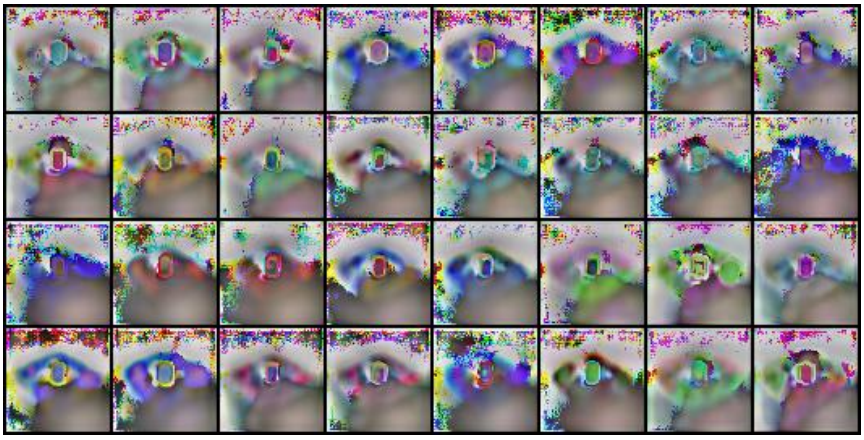
[H] 

Figure 12: DDPM w CFG new_test.json

```
accuracy of test.json on epoch 1:  0.153
accuracy of new_test.json on epoch 1:  0.131
```

Figure 13: DDPM w CFG Accuracy Screen Shot

4.3 LDM

For LDM, it seems to be too blur as the dataset is too small for unet to learn laten space input transform to image as the channel is 4.

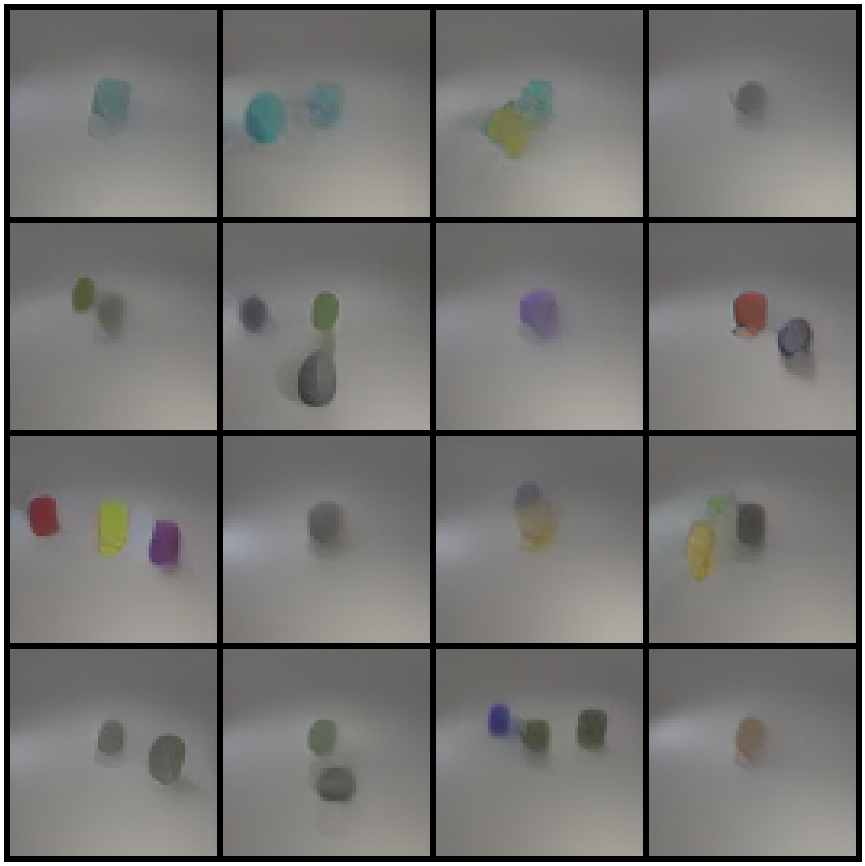


Figure 14: LDM test.json

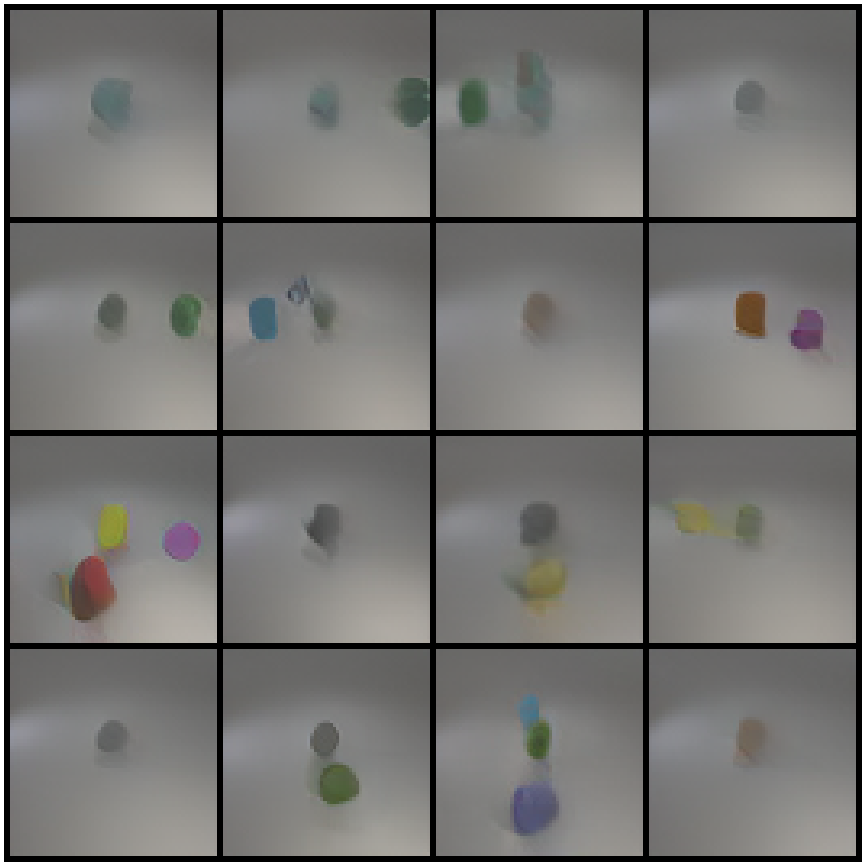


Figure 15: LDM new_test.json

```
accuracy of test.json on epoch 15:  0.278
accuracy of new_test.json on epoch 15:  0.262
```

Figure 16: LDM Accuracy Screen Shot